UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

DISEÑO E IMPLEMENTACIÓN DE UN BOT CONVERSACIONAL PARA COACHING SOBRE CIBERSEGURIDAD

Sonia Moreno Cano
Tutor: Álvaro Ortigosa Juárez

JUNIO 2020

DISEÑO E IMPLEMENTACIÓN DE UN BOT CONVERSACIONAL PARA COACHING SOBRE CIBERSEGURIDAD

AUTOR: Sonia Moreno Cano
TUTOR: Álvaro Ortigosa Juárez

Dpto. Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
junio de 2020

Resumen (castellano)

En la actualidad, la mayoría de las personas utilizamos recursos tecnológicos, estamos conectados gracias a Internet; y aparte de los beneficios que nos aportan estas herramientas, también existe el peligro de que se produzcan ataques de malware contra nosotros. Por ello, es necesario actuar de forma responsable, y estar informado sobre los posibles ataques, y diferentes medidas para evitar y solucionar estos problemas.

En este Trabajo de Fin de Grado se ha desarrollado un bot conversacional sobre ciberseguridad. Con el objetivo de poder ayudar a un público genérico a defender sus equipos contra ataques fraudulentos, informarles sobre cómo actuar ante un determinado ataque y cómo prevenir los mismos. En resumen, ofrecer apoyo y consejos a los usuarios para proteger sus equipos e información.

Se exponen las herramientas utilizadas para la creación del bot, comparándolas con otras y explicando sus ventajas e inconvenientes. También se explica cómo se ha ido desarrollando, los ficheros que lo componen, base de datos que se utilizan y los distintos entornos en los que se ha probado. El bot cuenta con una base de datos que se podrá ir actualizando con nuevos malwares que vayan surgiendo, pudiendo los usuarios preguntar por uno en concreto. También se podrán recoger las propuestas de los mismos usuarios, registrar cómo ellos solucionaron cierto problema, con el fin de recomendárselo a otros usuarios. Incluso analizar la efectividad de las soluciones propuestas, preguntando al usuario si le funcionó o no cierta solución. Esto ayudará a la hora de administrar la base de datos e ir poco a poco aportando un mejor servicio.

Abstract (English)

Today, most people use technological resources, we are connected thanks to the Internet, and apart from the benefits of these tools, there is also a danger of malware attacks against us. Therefore, it is necessary to act responsibly, and to be informed about possible attacks, and different measures to prevent and solve these problems.

A cybersecurity conversational bot has been developed in this Bachelor Thesis. In order to help a generic audience defend their computers against fraudulent attacks, inform them about how to act against a particular attack and how to prevent them. In short, offer support and advice to users to protect their computers and information.

The tools used to create the bot are exposed, comparing them with others and explaining their advantages and disadvantages. It also explains how it has been developed, the files that make up it, database that is used and the different environments in which it has been tested. The bot has a database that can be updated with new malwares that are emerging, with users being able to ask for a specific one. Proposals from the same users may also be collected, record how they fixed a certain problem, in order to recommend it to other users. Even analyze the effectiveness of the proposed solutions, asking the user whether or not a certain solution worked. This will help when managing the database and going slowly providing a better service.

Palabras clave

Ciberseguridad, prevención, protección, malware, robo de información, bots conversacionales.

Keywords

Cybersecurity, prevention, protection, malware, information theft, conversational bots.

Agradecimientos

A mi familia y amigos, gracias por apoyarme durante toda mi etapa académica.

INDICE DE CONTENIDOS

1 Introducción

1.1 Motivación

Hoy día, en la era de la tecnología, disponemos de un fácil acceso a grandes cantidades de información, existiendo una gran conectividad entre las personas. El problema de esta conectividad es que hay mayor riesgo a sufrir un ataque, por tanto, la seguridad es elemental para poder avanzar en nuestro proceso de digitalización.

En la actualidad, las grandes empresas y PYMES, entre otras organizaciones, se han convertido en el blanco perfecto para ciberdelincuentes, piratas informáticos y crackers. Recordemos, por ejemplo, el ransomware WannaCry, malware (es decir, software malicioso) que se esparció rápidamente por Internet y las redes locales. WannaCry consiguió inutilizar más de 200.000 ordenadores en 150 países. En algunos hospitales, WannaCry cifró todos los dispositivos, incluido el equipo médico, y algunas fábricas se vieron obligadas a detener su producción. Otro ejemplo es Stuxnet, un gusano capaz de expandirse en oculto a través de memorias USB y penetrar incluso en ordenadores que no estuvieran conectados a Internet o a una red local.

No solo las empresas deben tomar medidas concretas contra los ataques maliciosos, también los usuarios deben proteger sus equipos y estar informados sobre los posibles ataques que pueden sufrir sus equipos, o cómo deshacerse de los malwares. Para ello, se propuso el desarrollo de un bot conversacional (en adelante chatbot) capaz de resolver las dudas de ciberseguridad que les surjan a los usuarios, o alertar de los malwares más actuales y frecuentes. Se desarrolló con Python, utilizando una librería llamada Rasa, pudiéndose desplegar en distintas plataformas como Telegram, Slack, Facebook, entre otras.

Cabe destacar que los chatbots son capaces de resolver dudas de forma rápida, manejando una gran cantidad de datos. Así, el usuario puede obtener una respuesta instantánea, a diferencia de si la tuviera que buscar, por ejemplo, en una guía. También se puede realizar un análisis de datos, por ejemplo, examinando la utilidad de las respuestas propuestas por el chatbot, preguntando sugerencias a los usuarios o realizando un cómputo sobre los términos más demandados. Lo anterior es información bastante útil, pudiéndose utilizar para mejorar la calidad del servicio ofrecido.

1.2 Objetivos

El principal objetivo de este TFG es el desarrollo de un chatbot en Python utilizando la librería Rasa y la posterior puesta en producción en Telegram, con la finalidad de ayudar a los usuarios a proteger sus equipos y proporcionar información acerca de la ciberseguridad.

El chatbot muestra descripciones de ciertos tipos de malware y soluciones a los mismos. También cuenta con un menú de ayuda, e información general sobre ciberseguridad para responder a las dudas de los usuarios. Los administradores pueden añadir a la base de datos distintas soluciones a determinados problemas, también se puede tener en cuenta las propuestas de los usuarios, preguntándoles cómo eliminaron cierto malware. Además, si un

usuario pregunta sobre algún tema en concreto, se le muestran ciertas sugerencias con información relacionada.

Se registran los malwares citados por los usuarios con el fin de poder concluir cuáles son los más buscados en un momento determinado. Esto permite avisar a otros usuarios sobre algún virus actual que esté afectando a muchas personas.

Por último, también se ofrece un servicio de gestión de contraseñas, que permite crear modelos de contraseñas seguras con una longitud definida por el usuario y analizar si una contraseña introducida es lo suficientemente segura. Las contraseñas utilizadas en este servicio se crean con el fin de servir como ejemplos para el aprendizaje del usuario, o como modelos que podrán ser modificados para crear contraseñas reales, no para ser usadas directamente en las aplicaciones. Cabe destacar que si algún usuario utilizase una de las contraseñas generadas por este servicio, no se podría conocer la aplicación ni el usuario con el que la está utilizando, por lo que tampoco sería un riesgo de seguridad.

1.3 Organización de la memoria

En la segunda sección se incluye el estado del arte en ella se habla de las características del leguaje humano, el concepto de Procesamiento de lenguaje natural (NPL) y diferencias entre distintas plataformas para la implementación de chatbots.

A continuación, en la tercera sección, se expone el diseño utilizado, explicando qué es Rasa, y las bases de datos, visores de bases de datos y librerías que se han utilizado.

Después, en la cuarta sección, se explica el desarrollo realizado: la configuración, acciones y base de datos necesarias para crear el chatbot.

En la quinta sección, se comenta la integración con Telegram y las pruebas realizadas.

Finalmente, el capítulo final describe las conclusiones del trabajo.

2 Estado del arte

2.1 Lenguaje humano

Para poder desarrollar un chatbot es necesario contar con un sistema capaz de comprender e interpretar el lenguaje humano. Para un ordenador es complicado analizar el lenguaje humano debido a su ambigüedad, polisemia y variación del significado dependiendo del contexto.

Con el objetivo de poder crear un sistema capaz de entender el lenguaje humano, se debe tener en cuenta las distintas áreas de la lingüística [2]:

- Morfología: estudia las reglas que rigen la flexión, la composición y la derivación de las palabras.
- Sintaxis: estudia el orden y la relación de las palabras o sintagmas en la oración, así como las funciones que cumplen.
- Semántica: estudia el significado de las expresiones lingüísticas.
- Pragmática: estudia el lenguaje en su relación con los usuarios y las circunstancias de la comunicación.
- Fonología: estudia los fonemas o descripciones teóricas de los sonidos vocálicos y consonánticos que forman una lengua.

2.2 NPL

El Procesamiento de Lenguaje Natural (NPL por su sigla en inglés) es un campo de las ciencias de la computación, inteligencia artificial y lingüística que estudia las interacciones entre las computadoras y el lenguaje humano [3]. Ya sea con el fin de proponer una traducción entre dos idiomas, realizar una interpretación del lenguaje hablado, ejecutar comandos de voz, generar lenguaje hablado para facilitar la comunicación con personas invidentes o emitir respuestas automáticas a preguntas que realicen los usuarios.

En el caso de la traducción, por ejemplo, existe una gran probabilidad de cometer fallos si la máquina solo se centra en el significado de cada término por separado. Con una programación basada en NLP no surgiría este problema, pues se tendría en cuenta el significado de las secciones de texto que guardan relación entre sí, como modismos o frases hechas. De este modo se reduce la ambigüedad del lenguaje y se resuelven casos de polisemia.

Es interesante conocer a grandes rasgos cómo funciona NPL. Para responder a esta cuestión, en necesario dividir el proceso que realiza en diferentes partes:

- *Part-Of-Speech Tagging* o POST: está relacionado con la morfología, en esta parte se extrae el significado de cada palabra por separado. Para evitar la ambigüedad del lenguaje se utilizan corpus de texto o algoritmos de autoaprendizaje.

- *Parse trees*: se utilizan diagramas de análisis sintáctico para poder entender la estructura de las oraciones.

- Por último, se intenta determinar el significado de una palabra con ayuda de las palabras que le preceden o le siguen, con el fin de seleccionar el significado correcto ante una palaba polisémica.

NLP también ofrece la posibilidad de manejar grandes cantidades de datos de forma consistente e imparcial. Esta automatización es necesaria teniendo en cuenta la gran cantidad de datos que se generan diariamente, como, por ejemplo: registros médicos, medios sociales, etc.

En resumen, NLP es una rama de la inteligencia artificial que ayuda a las computadoras a entender, interpretar y manipular el lenguaje humano. Bastante útil debido a la ambigüedad y diversidad del lenguaje y a la gran cantidad de datos que se genera día a día.

2.3 Chatbots

A continuación, nos centraremos en el uso de NLP para chatbots, herramientas en las cuales se necesita hallar una respuesta ante la solicitud que ha enviado el usuario. Existen multitud de plataformas para chatbots que utilizan NLP, entre ellas, Rasa [32], Dialogflow [33], Bot Framework [34], Wit.ai [35] o Amazon Lex [36].

2.3.1 Dialogflow

Dialogflow es una herramienta creada por Google, permite la creación de chatbots o agentes, capaces de entender el lenguaje natural.

No requiere de ningún tipo de instalación y es sencillo de utilizar, sin ni siquiera necesitar una línea de código para crear un determinado chatbot. Así mismo, proporciona una consola para poder crear, compilar y probar agentes, y el código puede ser almacenado en la nube. Una de las desventajas de Dialogflow es que debido a su servicio de alojamiento en la nube impide al usuario el alojamiento en local o en su propio servidor.

2.3.2 Microsoft Bot Framework

Microsoft Bot Framework ofrece una serie de servicios y herramientas que permiten la construcción, publicación y despliegue de chatbots, existiendo dos versiones, una gratuita y otra versión de pago.

Ofrece una gran variedad de funciones con las que se puede llegar a construir asistentes virtuales sofisticados, además es flexible y escalable. Así como permitir la integración con multitud de plataformas como Slack [37], Facebook [38], Messenger [39], Telegram [40] o Skype [41]. También permite la creación de chatbots a partir de una serie existente de preguntas y respuestas frecuentes.

Una de las ventajas que ofrece es su compatibilidad con Azure Bot Service [42], debido a que Azure puede permitir la rápida respuesta a las preguntas de los usuarios, incluso si hay un gran volumen de tráfico. Otro beneficio es proporcionar un SDK de código abierto para construir y crear chatbots, también se puede utilizar la aplicación Microsoft Bot Framework Emulator [43] para su depuración.

2.3.3 Wit.ai

Wit.ai es utilizado para la creación de aplicaciones y dispositivos con los que se puede hablar o enviar mensajes de texto. Es de código abierto y también permite crear una interfaz de voz para sus aplicaciones. Se puede utilizar como asistente para el hogar, ya que puede controlar dispositivos inteligentes, incluidos electrodomésticos y dispositivos portátiles, como, por ejemplo, controlar la temperatura del hogar.

Wit.ai cuanta con un sistema NPL bastante completo: en contraposición, algunos usuarios percibieron cierta dificultad a la hora de entrenar el chatbot, o recuperar ciertos parámetros [10].

2.3.4 Amazon Lex

Amazon Lex utiliza el reconocimiento automático de voz (ASR) y la comprensión del lenguaje natural (NLU) para permitir a los desarrolladores crear chatbots fáciles de usar. Cuenta también con un aprendizaje automático AWS, con el cuál el chatbot podrá ir aprendiendo.

Amazon Lex ofrece una serie de ventajas, como la posibilidad de reconocer el lenguaje hablado y transformarlo en texto, es escalable y ofrece integración con móviles.

Como desventajas, no es multilenguaje, solo soporta inglés. Otro inconveniente es la dificultad a la hora de preparar set de datos de pruebas, pues el mapeo de expresiones y entidades es en cierta medida complicado.

2.3.5 Rasa

Rasa es un toolkit de código abierto para la implementación de AIs conversacionales.

Rasa cuenta con una documentación bastante completa e interactiva, además ofrece la oportunidad de poder personalizar el chatbot con el uso de Python. Desde el punto de vista de un desarrollador, su uso permite un aprendizaje más detallado acerca de cómo funciona un chatbot. Incluso, al ser open source, permite investigar o modificar el código base. También permite ejecución en local y añadir la implementación al propio servidor del desarrollador, opción no posible para las herramientas que solo utilizan la nube. Además, fue un requisito impuesto por el tutor, jugando el papel de cliente de proyecto de software, que se debía utilizar esta tecnología.

3 Diseño

3.1 Rasa

Como se definió anteriormente, Rasa es un Toolkit de código abierto para la implementación de AIs conversacionales. El repositorio se puede encontrar en la siguiente url: https://github.com/RasaHQ/rasa

3.1.1 Componentes

Rasa Stack se puede dividir en dos componentes:

- Rasa NLU: es la parte que se encarga de procesar el lenguaje natural para poder clasificarlo según la intención del usuario. Así, por ejemplo, la frase: "*¿Es mi contraseña segura?"* devolvería una estructura similar a ésta:


  ```
  {
    "intent": {
      "name": "request_strength_password",
      "confidence": 0.9977140426635742,
      "entities": [],
       "text": "¿Es mi contraseña segura?"
    }
  ```


  De esta manera, Rasa calcula la siguiente acción (*request_strength_password*) con una confidencialidad del 0.997. El campo *entities* se corresponde a propiedades cuyo valor puede obtenerse a través del texto que escriba el usuario. Por último, el campo *text* hace referencia al texto original que se escribió.

  Para poder observar cómo se procesa el texto que introduce el usuario se puede utilizar el comando *Rasa Shell nlu*. Esto nos puede ayudar a la hora de resolver ciertos errores, por ejemplo, cuando el chatbot realiza una acción no deseada en vez de la que se esperaba. También para comprender mejor el funcionamiento de esta librería.

- Rasa Core: se encarga de la gestión de los diálogos. Basándose en la entrada NLU, en el historial de conversaciones y en los datos de entrenamiento, es capaz de predecir la próxima acción del chatbot.

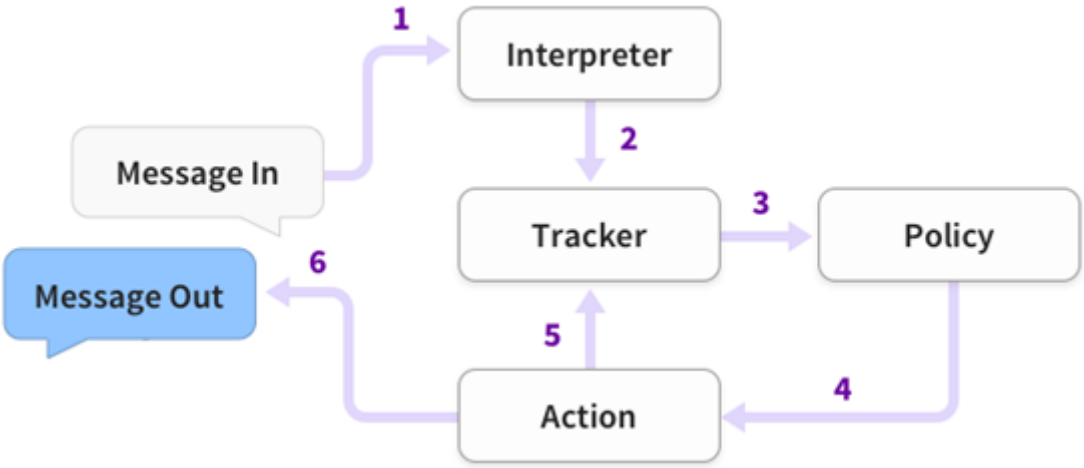


Imagen 1. Arquitectura Rasa CORE. Recuperado de chatbotslife.com

1- El mensaje es enviado por el usuario y se pasa al intérprete, éste lo convierte en un diccionario, incluyendo, el texto original, la intención y las entidades usadas.

2- El tracker realiza un seguimiento del estado de la conversación, y recibe la información de que se ha recibido un nuevo mensaje.

3- Se recibe el estado actual del tracker.

4- Policy escoge cuál será la siguiente acción.

5- El tracker registra la acción seleccionada.

6- Se le envía la respuesta al usuario.

3.2 Base de datos

3.2.1 SQLite

Es un sistema de gestión de base de datos relacional. El código de SQLite [44] es de dominio público y libre para cualquier uso, ya sea comercial o privado.

SQLite es embebido, es decir, no es un proceso independiente con el que el programa principal se comunica, a diferencia de como ocurre en una base de datos cliente-servidor. Así, la biblioteca SQLite se enlaza con el programa principal, pasando a ser parte integral del mismo. Esta arquitectura sin servidor permite que la base de datos sea compatible con varias plataformas. La base de datos está contenida dentro de un solo archivo de disco y todas las lecturas y escrituras tienen lugar directamente en él. A la hora de realizar transacciones, se acoge a las políticas ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad).

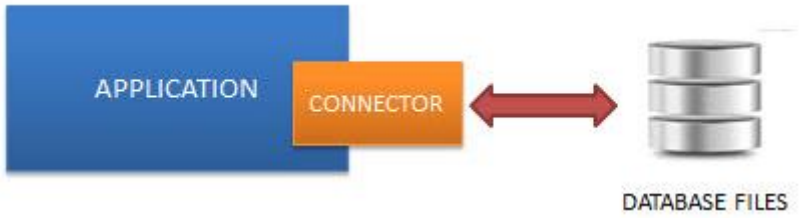


Imagen 2. Estructura de SQLite. Recuperado de tableplus.com

Ventajas

Es una librería bastante compacta, su tamaño puede ser fácilmente inferior a 600KB. Esto permite una mejor portabilidad y sencilla instalación. Gracias a su pequeño tamaño es muy adecuado para IOT y para dispositivos integrados.

Cuenta con funcionalidad más simple que PostgreSQL y más adaptada a casos más comunes. Al realizar llamadas sencillas a subrutinas y funciones se reduce la latencia en el acceso a la base de datos, y el proceso es bastante rápido.

También puede actuar como complemento para una base de datos cliente-servidor. Por ejemplo, puede almacenar en caché cierto tipo de datos con la finalidad de reducir la latencia de las consultas.

Desventajas

Uno de los principales inconvenientes del sistema SQLite es su falta de capacidades multiusuario. Esto se traduce en una falta de control de acceso y falta de seguridad, pues no cuenta con un sistema de autenticación; la base de datos puede ser leída o actualizada por cualquier usuario. Sería apropiado elegir bases de datos cliente-servidor como PostgreSQL en estos casos, especialmente cuando se trata de grandes conjuntos de datos como Big Data.

Otra desventaja es su limitación a las operaciones básicas, mientras que la fortaleza de un RDBMS avanzado como PostgreSQL es su extensibilidad con procedimientos almacenados.

También hay que tener en cuenta que SQLite solo soporta cinco tipos de datos: BLOB (binary large object), NULL, INTEGER, TEXT, REAL.

3.2.2 PostgreSQL

PostgreSQL [45] es un sistema de gestión de base de datos relacional orientado a objetos y de código abierto, que realiza un especial énfasis en la extensibilidad y en el cumplimiento de estándares. A diferencia de SQLite, utiliza un modelo cliente-servidor; contando con un proceso en la parte del servidor, que maneja las comunicaciones del cliente, y administra los archivos y operaciones de la base de datos. PostgreSQL mantiene también los principios ACID, lo que permite transacciones más confiables.

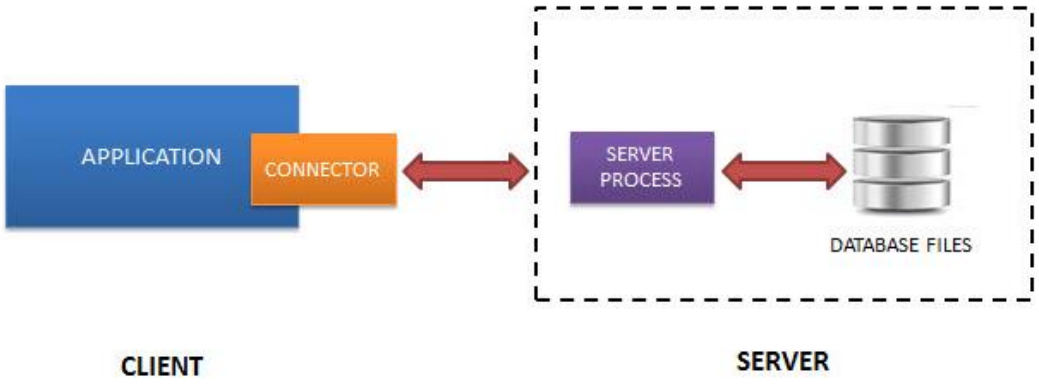


Imagen 3. Estructura de SqLite. Recuperado de tableplus.com

Ventajas

PostgreSQL ofrece mayor cantidad de operaciones avanzas que SQLite, y permite los procedimientos almacenados.

En cuanto a seguridad, cuenta con más funcionalidades y configuraciones para facilitarla. Así, proporciona un control de acceso de usuario basado en host y autenticación de usuario. PostgreSQL también suministra de forma nativa la capacidad de cifrar las comunicaciones cliente-servidor utilizando SSL. Por lo que ofrece mayor seguridad que SQLite.

También cuenta con un control de concurrencias multiversión o MVCC para mantener la consistencia de los datos durante el acceso concurrente de los mismos. Es una técnica de concurrencia donde ninguna tarea o hilo es bloqueado mientras se realiza una operación en la tabla, porque el otro hilo usa su propia copia o versión del objeto dentro de una transacción. Así se proporciona una imagen del estado de la base de datos en cada transacción. Esta tecnología es superior al uso de bloqueos para la concurrencia, ya que minimiza la contención de bloqueos en entornos de múltiples usuarios, mejorando significativamente el rendimiento. Cuanta también con Hot-Standby, lo que permite que los clientes realicen búsquedas mientras los servidores se encuentran en mantenimiento. Así, los accesos de lectura no se ven afectados ante tareas de mantenimiento o recuperación.

Otra ventaja es que PostgreSQL maneja sesiones concurrentes de clientes creando un nuevo proceso para cada conexión. Este proceso es independiente del proceso maestro, creándose y destruyéndose durante la vida útil de la conexión del cliente. Debido a esta capacidad de procesamiento en paralelo, PostgreSQL es capaz de ejecutar operaciones complicadas de forma eficiente y rápida, siendo bastante útil para procesos analíticos. La extensibilidad de la base de datos PostgreSQL también la convierte en un candidato perfecto para proyectos de investigación y científicos.

Desventajas

A la hora de realizar operaciones simples puede llegar a ser más ineficaz que SQLite, además, su instalación y portabilidad son más complejas.

3.2.1 Conclusión

En el proyecto realizado se ven dos tipos diferentes de almacenamiento:

- Almacenamiento de nombre de métodos, acciones, intentos de usuario y otro tipo de información utilizada por el chatbot y transparente al desarrollador.
- Almacenamiento de los datos con los que va a trabajar el chatbot, como descripciones y soluciones a los distintos problemas.

Para el primer tipo de almacenamiento Rasa utiliza SQLite por defecto. En este caso Rasa realiza consultas sencillas para poder obtener información, y no se espera que en un futuro aumente mucho su tamaño y complejidad de las consultas, por lo que se ha decidido seguir usando SQLite para este sistema.

Mientras que, en el segundo caso, sí se espera un aumento de datos. Incluso, en un futuro, para poder analizar más información acerca de los usuarios, se podría incluir mayor número de tablas y búsquedas más complejas, por lo que se ha decidido optar por PostgreSQL.

Así se consigue la robustez de PostgreSQL para las operaciones complejas y se reduce la latencia en las operaciones sencillas con SQLite.

3.3 Visores de base de datos

Para el desarrollo del proyecto se han utilizado los siguientes programas para poder observar y analizar las bases de datos:

3.3.1 DB Browser for SQLite [46]

Es una aplicación gratuita y de código abierto diseñada para facilitar la creación y administración de las bases de datos con SQLite. Se puede utilizar tanto para Windows, Linux y MacOS.

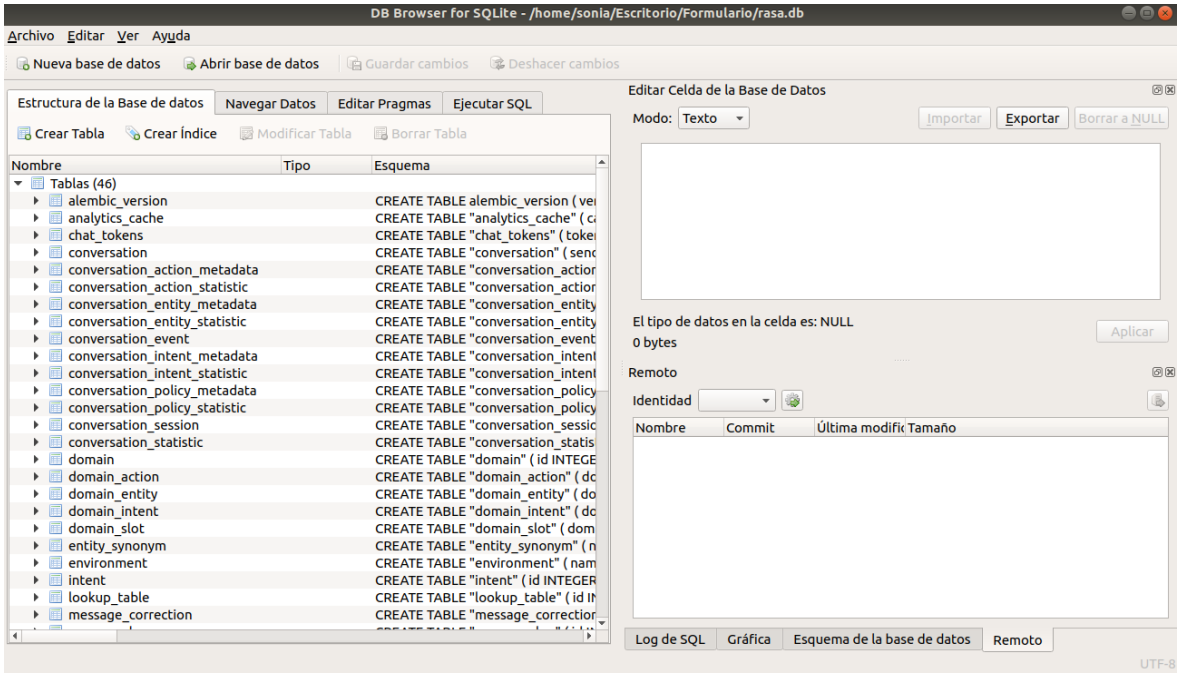


Imagen 4. DB Browser for SQLite. Captura de pantalla

3.3.2 PgAdmin III [47]

Es una herramienta de código abierto para la administración de bases de datos PostgreSQL. Se encuentra disponible para varios sistemas operativos, incluyendo: Microsoft Windows, Linux, FreeBSD, Mac OSX y Solaris.

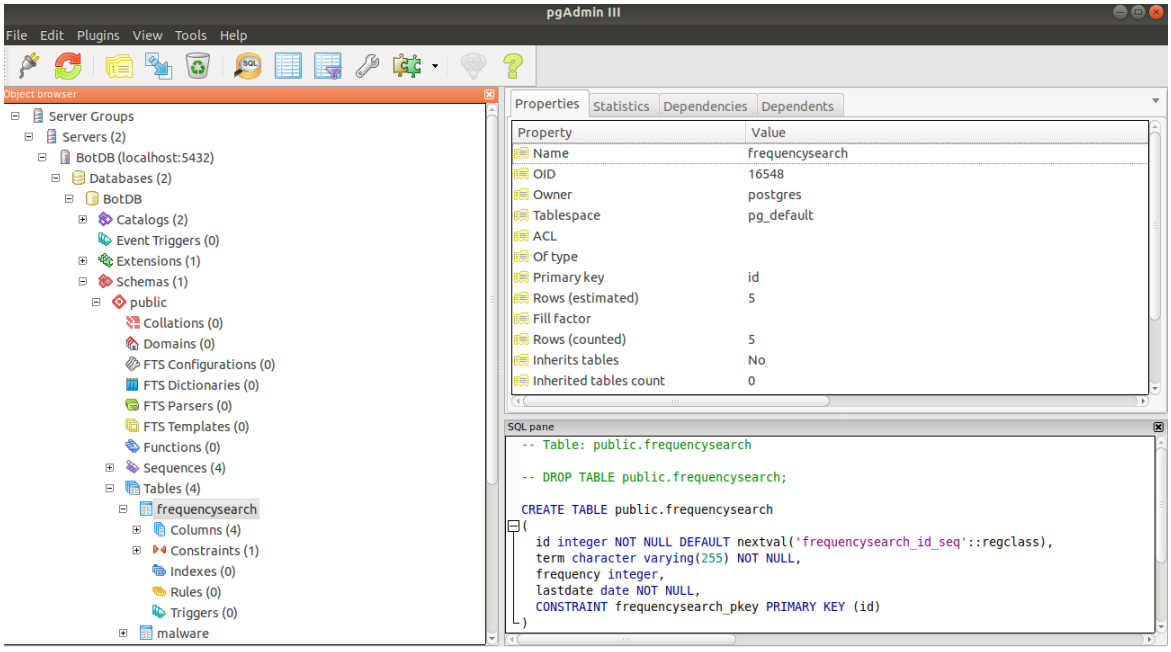


Imagen 5. PgAdmin III. Captura de pantalla

3.4 Librerías utilizadas

A continuación, se describen las librerías utilizadas para la gestión de base de datos.

3.4.1 Librerías Python para PosgreSQL

Para la implementación de este proyecto se ha utilizado la librería Psycopg2 [48] para poder acceder y realizar consultas en PosgreSQL desde el lenguaje Python.

Psycopg2 es una de las librerías más utilizadas en Python para PostgreSQL, principalmente por estos motivos:

- Es requerida para la mayoría de frameworks de Python y PostgreSQL.
- Realiza el mantenimiento de las versiones principales de Python, es decir, Python 3 y Python 2.
- Fue diseñado para aplicaciones multiproceso. Además, cuenta con Thread-safety o seguridad en hilos, en el cual se manipula las estructuras de datos compartidos de manera que se garantiza que todos los subprocesos se comporten correctamente y cumplan con sus especificaciones de diseño.

Existen otras librerías como: pg8000 [49], py-postgresql [50], PyGreSQL [51] o SQLAlchemy [52].

4 Desarrollo

4.1 Configuración de escenarios

La configuración de escenarios que pueden ocurrir en la conversación, así como la definición de acciones y eventos que puede realizar el chatbot vienen indicados en los siguientes ficheros.

A continuación se proporciona un esquema con los componentes que forman el proyecto.

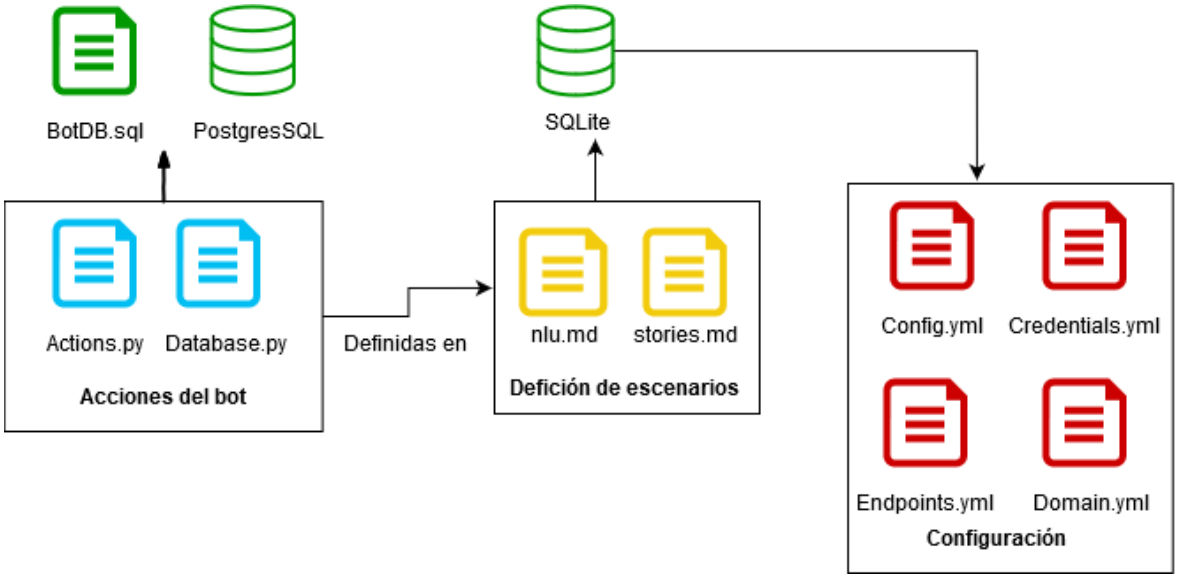


Imagen 6. Esquema de componentes.

4.1.1 Nlu.md

Forma parte de NLU. En este fichero se puede encontrar una serie de *intents* o intenciones. Como, por ejemplo:

```
## intent:greet
- Hola
- Hola, qué tal?
- qué tal?
- Buenas
- Buen día
- Buenos días
- Buenas tardes
- Buenas noches
```

Imagen 7. Fichero Nlu.md. Captura de pantalla.

Cada uno de estos *intent* hace referencia a una posible acción de usuario. Así, si el usuario escribe “Hola”, este texto será interpretado como la acción *greet*.

Se podrían utilizar tanto el formato Markdown como JSON para poder definir los diversos casos. En el proyecto, se ha utilizado Markdown debido a su mayor sencillez y facilidad de comprensión.

¿Cuándo sería aconsejable utilizable JSON? Este formato se podría utilizar en situaciones en las que haya un equipo de gran tamaño añadiendo datos de prueba. En ese caso, una posible propuesta sería crear una interfaz de usuario que envíe peticiones JSON al servidor.

Un ejemplo con JSON sería el siguiente:

```
{
   "rasa_nlu_data": {
      "common_examples": [],
      "regex_features": [],
      "lookup_tables": [],
      "entity_synonyms": []
   }
}
```

Donde *common_examples* hace referencia a los componentes *text*, *intent* y *enitites. Regex_features* son las expresiones regulares, entity*_synonyms* para los sinónimos y *lookup_tables* para las tablas de búsqueda.

En el fichero Nlu.md podemos encontrar distintos componentes:

- **Common examples:** cuenta con tres partes.
    - text o texto: mensaje que escribe el usuario.
    - intent: intención asociada al texto del usuario.
    - entities o entidades: partes del texto que necesitan ser identificadas.

En el ejemplo siguiente, el *intent* se correspondería con *request_solution_malwaretype*. El texto, con "¿Cómo elimino un Ransamoware?". Las *entities* serían "Ransomware", "Spyware" y "Troyano", todas ellas se identificarían como *name_malwaretype*.

intent:request_solution_malwaretype
- ¿Cómo elimino un [Ransomware](name_malwaretype)?
- ¿Cómo elimino un [Spyware](name_malwaretype)?
- ¿Cómo elimino un [Troyano](name_malwaretype)?

- Expresiones regulares: Pueden ayudar a la hora de clasificar mensajes que sigan algún patrón concreto. Por ejemplo: un código postal (cinco números).

regex:zipcode
- [0-9]{5}

- **Tablas de búsqueda:** Al igual que las *entities*, proporcionan una asociación de elementos. Las entidades candidatas para crear tablas de búsqueda son aquellas con un alcance definido.

lookup: name_malwaretype
data/test/lookup_tables/name_malwaretype.txt

Contenido del fichero name_malwaretype.txt:
Troyano
Spyware
Ransomware

En este caso, debido a la gran diversidad de tipos o nombres de *malwares* que puede buscar el usuario, se decidió utilizar *entities.*

- Sinónimos: Se utilizan para agrupar entidades diferentes, pero con el mismo valor.

##intent:search
- in the center of [NYC](city:New York City)
- in the centre of [New York City](city)

También se podría escribir de la siguiente manera:

synonym:New York City

- NYC

- nyc

De este modo, para el intérprete ambas cadenas de texto serían idénticas.

4.1.2 Stories.md

En este fichero se exponen los distintos escenarios que pueden ocurrir. Cada escenario cuenta con un nombre (*Greet deny user model*), con una serie de intentos por parte del usuario (*request_formulario* y *deny*), y las acciones que debe ejecutar el chatbot.

```
## Greet deny user model
* request_formulario
    - utter_can_make_questions
* deny
    - utter_sorry
    - action_chat_restart
```

Imagen 8. Fichero Stories.md. Captura de pantalla

Así, en el ejemplo anterior, tras el *intent* o intento *resquest_formurio,* el chatbot ejecutará la acción *utter_can_make_questions,* después el usuario proseguirá con la conversación.

A la hora de ejecutar el programa, el escenario planteado podría desenvolverse de la siguiente manera:

```
Your input ->  Realizar formulario
¿Puedo hacerte unas preguntas sobre tu equipo? Esto me ayudará a poder aconsejar
te mejor en un futuro.
Your input ->  No
De acuerdo, disculpa.
```

Imagen 9. Ejecución en servidor local. Captura de pantalla

4.1.3 Domain.yml

En este fichero se especifican los intentos, entidades, espacios y acciones que el chatbot conoce y maneja.

Así se encuentra dividido en varias partes:

- **intents:** se indican las intenciones definidas en el fichero nlu.md, representan las posibles peticiones que puede hacer el usuario. Un fragmento de esta lista sería:

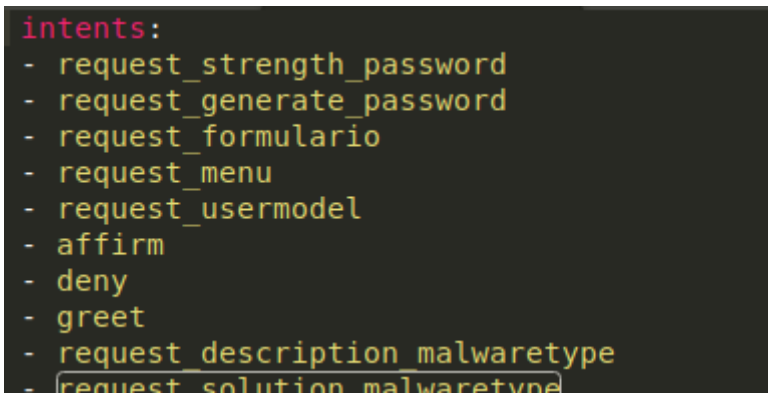


Imagen 10. Fichero Domain.yml, intents. Captura de pantalla

- **entities:** variables que se van modificando conforme avanza el diálogo entre el usuario y el chatbot. El valor de las mismas se puede almacenar gracias a los slots.

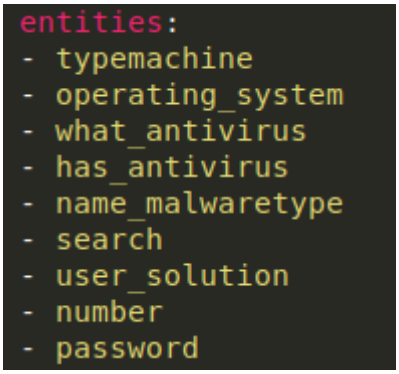


Imagen 11. Fichero Domain.yml, entities. Captura de pantalla

- **slots:** se utilizan para almacenar información proporcionada por el usuario. Cada slot definido en *domain.yml* consiste en una clave a la cual se le asignará posteriormente un valor determinado.

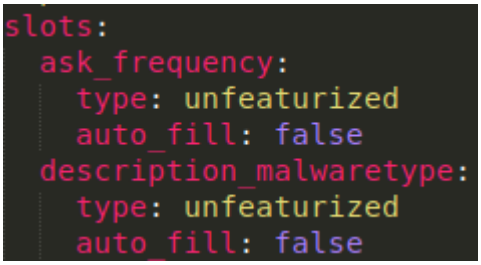


Imagen 12. Fichero Domain.yml, slots. Captura de pantalla

Existen distintos tipos de slots: *text*, *categorical*, *bool*, *float*, *list* o *unfeaturized*, este último utilizado cuando no se requiere que los datos almacenados afecten al flujo de la conversación. En el ejemplo de la imagen se iguala la propiedad *auto_fill* a *false* para evitar que estos slots puedan ser rellenados de forma automática a la hora de configurar los modelos NLU.

- **templates:** en este apartado se encuentran mensajes que le aparecerán al usuario. Pueden crearse mensajes estáticos, como los siguientes:

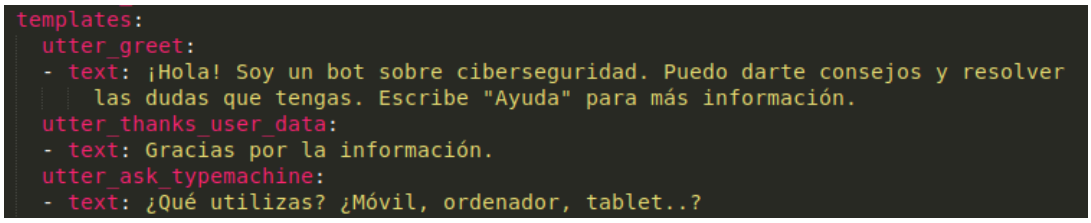


Imagen 13. Fichero Domain.yml, templates. Captura de pantalla

O dinámicos, dependientes del valor de alguna entidad o *entity*, que puede variar dependiendo de la situación que se dé.

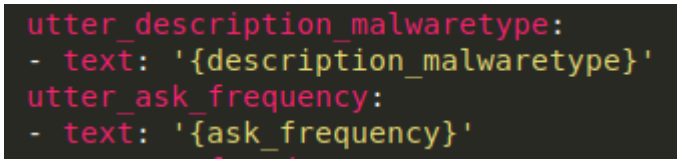


Imagen 14. Fichero Domain.yml, templates. Captura de pantalla

También se pueden añadir ciertos elementos, como botones a los que se les asigna una acción (*payload*). Incluso se pueden añadir imágenes con la etiqueta *image*.

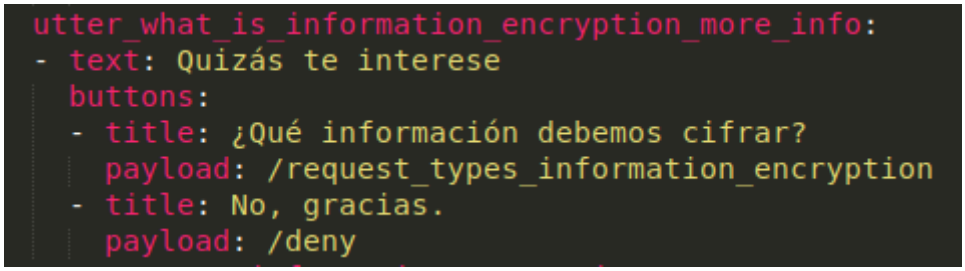


Imagen 15. Fichero Domain.yml, templates. Captura de pantalla

- **actions:** acciones ejecutadas por el chatbot como respuesta al mensaje del usuario.

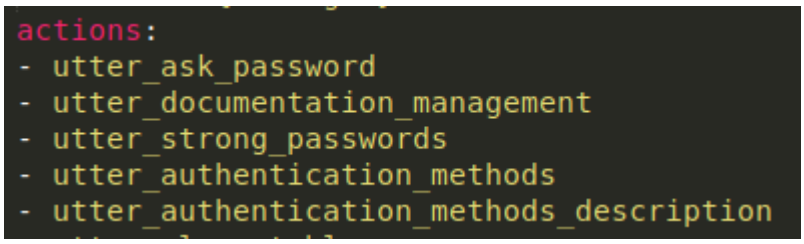


Imagen 16. Fichero Domain.yml, actions. Captura de pantalla

- **forms:** formularios utilizados por el chatbot.

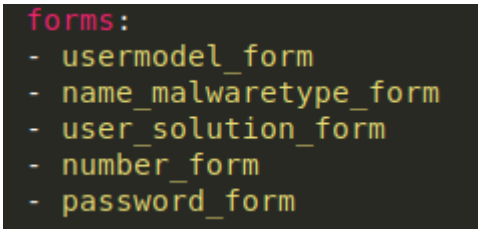


Imagen 17. Fichero Domain.yml, forms. Captura de pantalla

4.1.4 Config.yml

En este fichero se encuentra la configuración del NLU. Nos podemos encontrar diversos elementos como:

- **language:** utilizado para especificar el lenguaje del chatbot.
- **pipeline:** define el conjunto de componentes con los que se procesarán de manera secuencial los mensajes enviados por el usuario.
- **policies:** decide qué acción tomar en cada paso de la conversación.
  En este caso se han utilizado:
  - **MemoizationPolicy:** memoriza los datos de entrenamiento para poder predecir acciones.
  - **FormPolicy:** es una extensión de MemoizationPolicy, se encarga de predecir las acciones necesarias hasta que se completen todos los datos de los formularios.
  - **MappingPolicy:** se utiliza para asignar directamente intenciones a acciones.
  - **FallbackPolicy:** para poder establecer una acción por defecto (*fallback_action_name*), que se ejecuta si tiene una probabilidad de predicción por debajo de lo definido en *nlu_threshold*, o si las anteriores políticas no han podido concluir ninguna acción.

```yaml
policies:
  - name: MemoizationPolicy
  - name: FormPolicy
  - name: MappingPolicy
  - name: FallbackPolicy
    nlu_threshold: 0.01
    core_threshold: 0.01
    fallback_action_name: "action_default_fallback"
```

Imagen 18. Fichero Config.yml, policies. Captura de pantalla

4.1.5 Credentials.yml

Contiene las credenciales para las plataformas de voz y chat en las que se desplegará el chatbot.

La configuración para local es la siguiente:

```yaml
rasa:
  url: "http://localhost:5002/api"
```

Imagen 19. Fichero Credentials.yml, configuración local. Captura de pantalla

4.1.6 Endpoints.yml

Contiene los distintos *endpoints* que se pueden utilizar.

```yaml
action_endpoint:
  url: "http://localhost:5055/webhook"
```

Imagen 20. Fichero Endpoints.yml. Captura de pantalla

4.2 Acciones

Las acciones son las posibles respuestas que el chatbot puede dar en respuesta al texto que ha introducido el usuario. Existen varios tipos:

- **Utterance actions:** empiezan por *utter_*. Son cadenas de texto que se mostrarán al usuario. Este tipo de acciones se encuentran definidas en el fichero *domain.yml*.

```yaml
templates:
  utter_greet:
  - text: ¡Hola! Soy un bot sobre ciberseguridad. Puedo darte consejos y resolver
      las dudas que tengas. Escribe "Ayuda" para más información.
  utter_thanks_user_data:
  - text: Gracias por la información.
```

Imagen 21. Fichero Domain.yml. Captura de pantalla

- **Default actions:** son las que ya vienen previamente definidas por la librería de Rasa, como, por ejemplo: *action_listen***,** *action_restart***,** *action_default_fallback***.**

- **Retrieval actions**: empiezan por *respond_* y se suelen utilizan para agrupar intentos similares en un solo escenario.

- **Custom actions:** a este grupo pertenecen el resto de acciones, y pueden ejecutar diversos tipos de código. Este tipo de acciones se encuentran definidas en el fichero *Actions.py*.

  En el fichero Actions.py del proyecto, se pueden dividir en dos grupos diferentes de custom actions:

- **Acciones con formulario**: controlan los formularios en los cuales se le solicitará información al usuario. Cada acción representa una clase, y hereda de *FormAction*. Constan de tres métodos principales.
  - *name*: nombre de la acción.
  - *required_slots*: slots necesarios para completar la acción.
  - *slot_mappings*: especifica cómo se van a rellenar los slots.
  - *submit*: una vez se han proporcionado todos los slots requeridos, se procede a ejecutar la acción.

```
class UserSolutionForm(FormAction):
    def name(self) -> Text:
        """Unique identifier of the form"""
        return "user_solution_form"

    @staticmethod
    def required_slots(tracker: Tracker) -> List[Text]:
        return ["user_solution"]

    def slot_mappings(self) -> Dict[Text, Union[Dict, List[Dict]]]:
        return {
            "user_solution": [self.from_entity(entity="user_solution"), self.from_text() ],
        }

    def submit(
        self,
        dispatcher: CollectingDispatcher,
        tracker: Tracker,
        domain: Dict[Text, Any],
    ) -> List[Dict]:

        SaveUserSolution(tracker.get_slot('user_solution'), nameMalware)
        return []
```

Imagen 21. Fichero Actions.py. Captura de pantalla

- **Acciones sin formulario:** heredan de la clase *Action* y toman como métodos: *name*, para especificar el nombre de la acción y *run*, código que se ejecutará para llevarla a cabo.

```python
class AskTermsWithMoreFrequency(Action):
    def name(self):
        return 'action_ask_frequency'

    def run(self, dispatcher, tracker, domain):

        info = TermsWithMoreFrequency()

        if len(info) > 0:
            result = "Estos son los últimos términos más buscados: " + info[0]
            for i in info[1:]:
                result += ", " + i
            result += "."
            return [SlotSet('ask_frequency', result)]
        else:
            return [SlotSet('ask_frequency', "Disculpa, no encontre resultados.")]
```

Imagen 22. Fichero Actions.py. Captura de pantalla

Los nombres de estas acciones, en los ejemplos anteriores, *action_ask_frequency* y *user_solution_form*, son utilizados en el fichero *stories.md* para el desarrollo de los escenarios.

```
## Búsquedas más frecuentes
* request_ask_frequency
    - action_ask_frequency
    - utter_ask_frequency
    - action_chat_restart
```

Imagen 23. Fichero Stories.md. Captura de pantalla

Estas acciones también se definen en el fichero *domail.yml* para que el programa pueda reconocerlas.

Las acciones que se han creado son las siguientes:

- **UserModelForm:** se encarga de recoger información acerca del usuario: su sistema operativo, si tiene un antivirus activado, si se encuentra en móvil, pc, etc.

- **ChangeSlotSolution:** se utiliza para poder reutilizar código, pudiéndose devolver en *DescriptionMalwareType* el atributo que se deseé.

- **DescriptionMalwareType:** realiza una búsqueda en la base de datos para extraer la descripción de un determinado malware o una posible solución.

- **UserSolutionForm:** formulario para registrar una solución propuesta por el usuario, almacena la información en base de datos.

- **WorksIncrement:** se ejecuta si al usuario le funcionó la solución propuesta y lo registra en la base de datos.

- **NotWorkIncrement:** función análoga a la anterior, se ejecuta si la solución propuesta no ha sido acertada.

- **SuggestByOPeratingSystemAction:** el sistema sugiere al usuario posibles alertas que le puedan interesar según su sistema operativo.

- **AddUserAsk:** permite llevar un registro de los términos más buscados por los usuarios.

- **ActionRestarted:** se ejecuta al final de un escenario reseteando la conversación entre el usuario y el chatbot. Esto permite la ejecución de múltiples escenarios a lo largo de la sesión.

- **AskTermsWithMoreFrequency**: devuelve los términos cuya información ha sido más solicitada por los usuarios desde una fecha definida hasta la actual.

- **RandomSolutionUser**: devuelve una solución propuesta por los usuarios de forma aleatoria.

- **GetNewPasswordForm**: genera una contraseña nueva y segura, de una longitud especificada por el usuario.

- **CalculateStrengthPassword:** comunica al usuario de las debilidades de su contraseña, por ejemplo, si no está utilizando números o mayúsculas, o tiene una longitud pequeña.

4.3 Base de datos

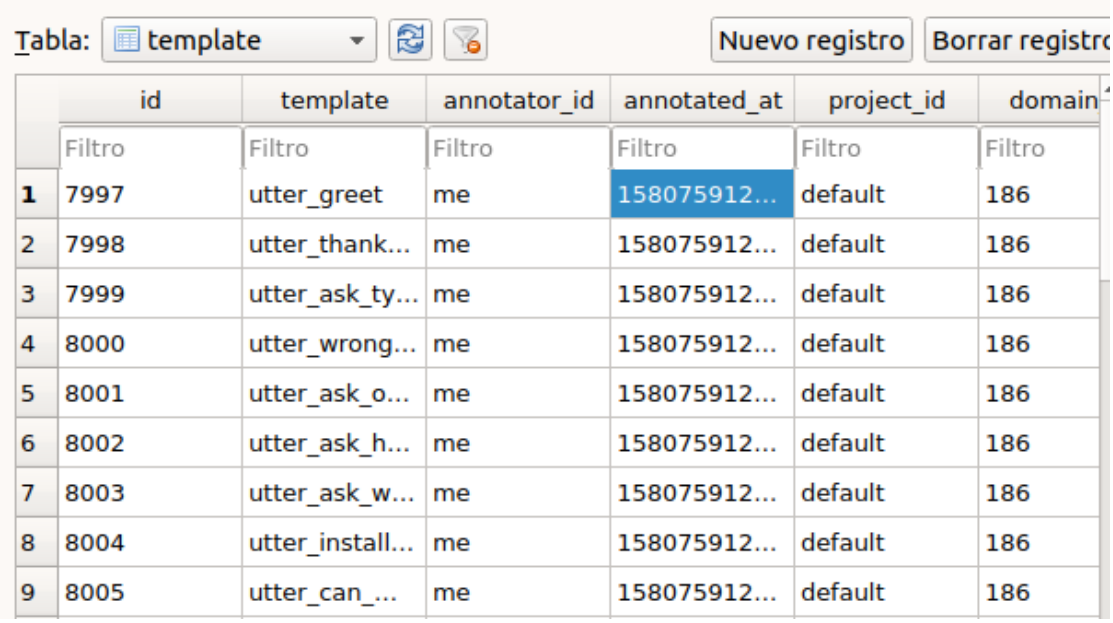
4.3.1 SQLite

Para almacenar las características y acciones de las que dispone el chatbot, se utiliza el fichero *rasa.db*.

En esta base de datos se incluyen tablas como *story*, en la cual se almacenan los escenarios definidos en *stories.md*.

Imagen 24. Base de datos SQLite. Captura de pantalla

Otra tabla interesante es la tabla *template*, muestra las posibles respuestas que puede dar el chatbot. Estas *templates* se encuentran en el fichero *domain.yml*.



	id	template	annotator_id	annotated_at	project_id	domain
	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro
1	7997	utter_greet	me	158075912...	default	186
2	7998	utter_thank...	me	158075912...	default	186
3	7999	utter_ask_ty...	me	158075912...	default	186
4	8000	utter_wrong...	me	158075912...	default	186
5	8001	utter_ask_o...	me	158075912...	default	186
6	8002	utter_ask_h...	me	158075912...	default	186
7	8003	utter_ask_w...	me	158075912...	default	186
8	8004	utter_install...	me	158075912...	default	186
9	8005	utter_can_...	me	158075912...	default	186

Imagen 25. Base de datos SQLite. Captura de pantalla

Otro ejemplo es *conversation_event*, donde se almacena la lista de eventos que se han ido ejecutando, el texto del usuario y la probabilidad de acierto.

En total se incluyen 46 tablas, las cuales contienen diversa información para el funcionamiento del chatbot.

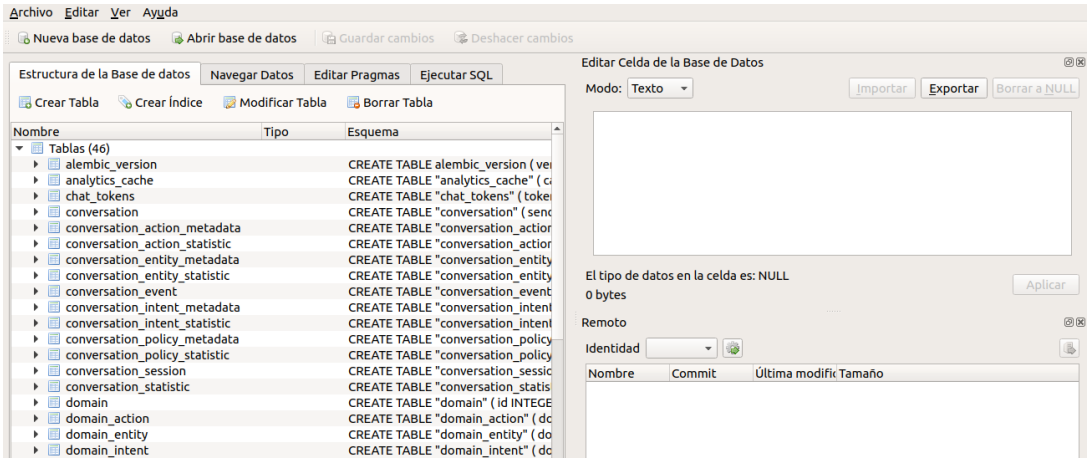


Imagen 26. Base de datos SQLite. Captura de pantalla

4.3.2 PostgreSQL

La estructura de las tablas es la siguiente:

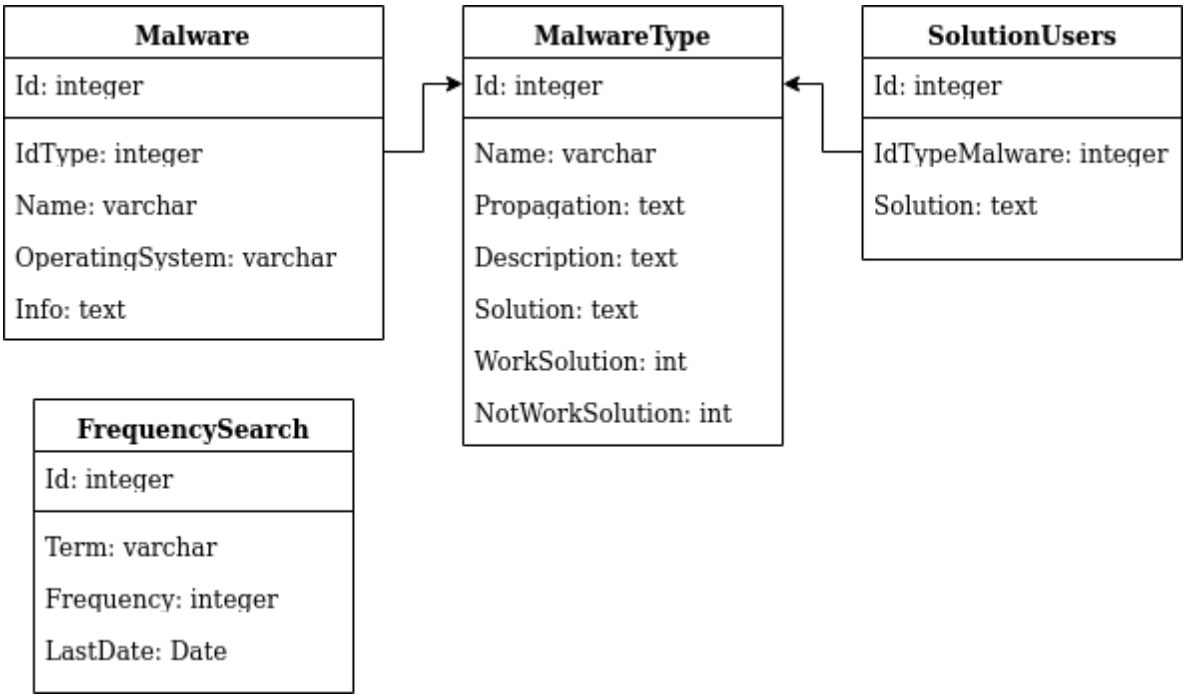


Imagen 27. Diagrama de base de datos utilizado en el proyecto

A continuación, se explica la funcionalidad de cada tabla:

- **Malware:** se almacenará la información sobre distintos malware: nombre, sistema operativo al que afecta y una breve descripción. Como, por ejemplo, malwares como *WannaCry*.

- **MalwareType:** en esta tabla se almacenan los distintos tipos de malware, de forma más genérica que en la tabla *Malware*. Se guarda el nombre y una descripción, también el tipo de máquina o sistema operativo por el que se propaga, una posible solución para eliminarlo, y el número de usuarios a los que le funcionó o no esa solución. Así se puede analizar la efectividad de la misma. Por ejemplo, en esta tabla se podría añadir un tipo de malware llamado *ransomware*, siendo *WannaCry* (añadido en la tabla anterior) un ejemplo de este tipo.

  Al realizar esta división de información entre las tablas *Malware* y *MalwareType* logramos que no se repita tanta información.

- **SolutionUsers:** se almacenan las soluciones propuestas por los usuarios. Si la solución propuesta en la tabla de *MalwareType* no es de utilidad para cierto usuario, se le puede proponer alguna de esta misma tabla, o incluso realizar un análisis para mejorar las soluciones propuestas en *MalwareType*.

- **FrequencySearch:** aparece la frecuencia de términos buscados. Gracias a esta información se puede añadir más términos a las tablas *Malware* y *MalwareType* basándose en las preferencias de los usuarios a la hora de solicitar información, o alertar de los virus más buscados actualmente.

En el fichero *Database.py* es donde se almacenan todas las consultas a la base de datos, así se registran diferentes métodos que son llamados por las acciones del chatbot definidas en *Actions.py*. Como ya se mencionó en el apartado de diseño, para realizar las posibles consultas se ha utilizado la librería *psycopg2*.

A la hora de realizar cada consulta hay que tener en cuenta una serie de pasos:

- Realizar la conexión de datos a través del método *connect*. Ya que a la hora de utilizar PosgreSQL se necesita una previa autentificación.

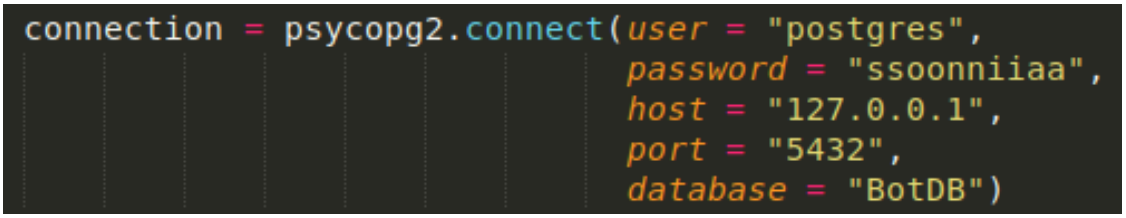


Imagen 28. Fichero Database.py. Captura de pantalla

  Como se puede observar, el usuario postgres, con su contraseña, va a acceder al servidor de la base de datos BotDB, levantado en: 127.0.0.1:5432.

- Se abre un cursor, necesario para realizar las diferentes operaciones.
  cursor = connection.cursor()

- Se crea una cadena con la correspondiente consulta. En este caso, se tomará el nombre del malware que es pasado como argumento a la función.
  postgreSQL_select_Query = "select description from malwaretype where name = %s"

- Ejecución de la consulta. *MalwareType* es el parámetro pasado a la función, que será transformado a *string(%s)* tal y como se ha especificado en paso anterior.
  cursor.execute(postgreSQL_select_Query, (malwareType,))

- Obtención del resultado:
  request = cursor.fetchall()

- En la consulta que se ha utilizado como ejemplo, no produce un cambio en la base de datos, pues se trata exclusivamente de un proceso de lectura. Para las operaciones que requieran de escritura, ya sea insertar, modificar o eliminar algún dato, es necesario realizar un *commit*, para ello se utiliza lo siguiente:
  connection.commit()

- Posteriormente se realizan las operaciones que se estimen necesarias con el resultado, por ejemplo, comprobar que no esté vacío, o tomar un valor arbitrario.

- Por último, la función devuelve el resultado obtenido.

También, es necesario tener en cuenta que a la hora de realizar este tipo de operaciones se puede producir ciertas excepciones, por ello es necesario utilizar *try*, *except* y *finally*. En try se realiza todo el procedimiento anterior, en *except* se muestra un aviso de error y en *finally* se procede a cerrar el cursor y la conexión.

Un ejemplo de una función completa sería:

```
def getMalwareTypeSolution(malwareType):
    try:
        connection = psycopg2.connect(user = "postgres",
                                      password = "ssoonniiaa",
                                      host = "127.0.0.1",
                                      port = "5432",
                                      database = "BotDB")
        cursor = connection.cursor()
        postgreSQL_select_Query = "select solution from malwaretype where name = %s"
        cursor.execute(postgreSQL_select_Query, (malwareType, ))
        request = cursor.fetchall()
        AddSearch(malwareType)
        if len(request) > 0:
            return  request[0]
        else:
            return ""

    except (Exception, psycopg2.Error) as error:
            print("Error fetching data from PostgreSQL table", error)

    finally:
        # closing database connection
        if ('connection' in locals()):
            cursor.close()
            connection.close()
            print("PostgreSQL connection is closed \n")
```

Imagen 29. Fichero Database.py. Captura de pantalla

Las funciones que se han creado para realizar operaciones en la base de datos son las siguientes:

- **GetMalwareTypeDescription:** devuelve la descripción de un determinado malware en base de datos. Toma como argumento el nombre del malware a buscar.

- **GetMalwareTypeSolution:** es parecido al anterior, solo que devuelve una posible solución en vez de la descripción.

- **SaveUserSolution:** añade una nueva solución sugerida por el usuario. Toma como argumentos el nombre del malware y la solución propuesta.

- **IncrementWorks:** es llamada si al usuario le ha resultado útil la solución propuesta por el chatbot. Modifica la tabla Malwaretype sumándole una unidad al atributo WorkSolution.

- **IncrementNotWork:** realiza la función análoga a la anterior. Suma una unidad al atributo NotWorkSolution de la misma tabla.

- **SuggestByOperatingSystem:** busca información acerca de posibles alertas que pudieran interesar al usuario basándose en su sistema operativo. Toma como argumento el sistema operativo del usuario.

- **AddSearch:** modifica la tabla *FrequencySearch*, añadiendo una unidad a la frecuencia de un determinado término que es pasado como argumento. Esta función es llamada cuando el usuario pide información acerca de un determinado término.

- **TermsWithMoreFrequency:** permite al usuario conocer los términos más buscados recientemente. Esto puede servirle como aviso sobre los problemas actuales que están teniendo otros usuarios.

- **RandomSolutionUserDB:** toma como argumento el nombre de un determinado malware, y devuelve una solución aleatoria proporcionada por otro usuario.

5 Integración, pruebas y resultados

5.1 Integración con Telegram

Rasa permite la integración de forma sencilla con distintas plataformas, como, por ejemplo, Facebook, Slack, Twilio, Google Hangouts Chat o Telegram. La integración en todas estas plataformas es similar, en este caso se ha utilizado Telegram, pero podría haberse usado cualquier otra.

Para poder realizar esta subida, es necesaria una serie de pasos.

5.1.1 Configuración de url con Ngrok

Ngrok [53] es una herramienta que permite acceder nuestro servidor local a cualquier persona en Internet con la que compartamos una url generada dinámicamente. Para poder obtener esta url, necesitamos seguir una serie de pasos:

- En la terminal ejecutar el siguiente comando. En esta ocasión se ha elegido utilizar el puerto 5005.

  ./ngrok http 5005

- Actualización del fichero *credentials.yml*, con la nueva información. *Access_token* y *verify,* son parámetros proporcionados por Telegram a la hora de crear el chatbot. Mientras que *Webhook_url* se forma con la url proporcionada por Ngrock añadiéndole */webhooks/telegram/webhook.*

  telegram:
  access_token:"971497924:AAHhUDZnfJjIU8so7xMId2c-Mqg-d4BwQOY"
  verify: "prueba117bot"
  webhook_url: https://c8340451.ngrok.io/webhooks/telegram/webhook

5.1.2 Conexión a la base de datos PostgreSQL

A la hora de realizar la conexión a la base de datos (*Database.py*), se debe indicar el usuario, contraseña, nombre de la base de datos y puerto. La creación de las tablas y los *inserts* se encuentran en *BotDB.sql*.

5.1.3 Despliegue

Para este paso se proporciona un Makefile. Se debe ejecutar los siguientes comandos en terminales diferentes:

make action-server
make telegram

5.2 Pruebas y resultados

Durante la conversación con el chatbot nos podemos encontrar distintos tipos de mensajes, como los siguientes:

- Saludo inicial al usuario, se proporciona también un menú para poder acceder a las funcionalidades del chatbot de forma más directa:

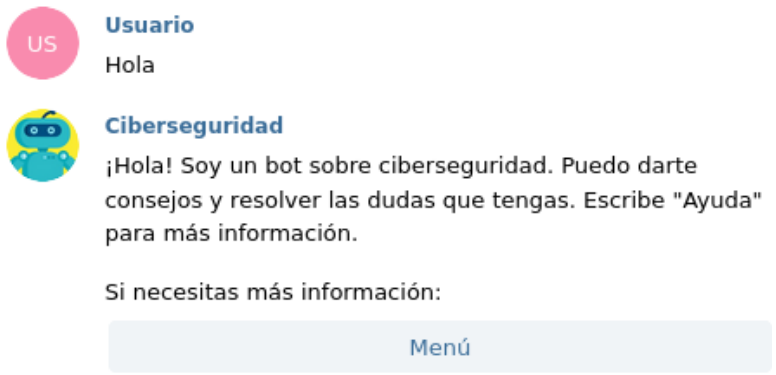


Imagen 30. Prueba con Telegram. Captura de pantalla.

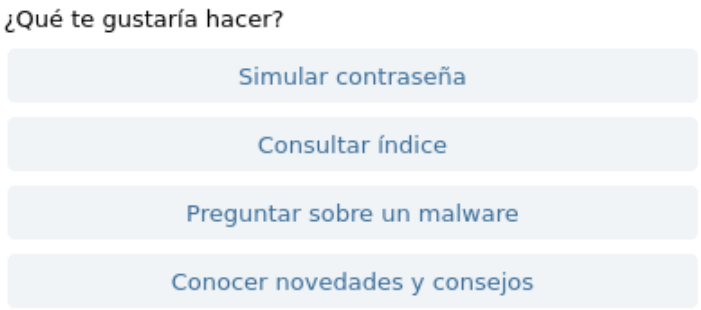


Imagen 31. Opciones del menú. Captura de pantalla.

- Información para ayudar al usuario. Si el usuario escribe un mensaje no reconocido por el chatbot, se le sugerirá acceder a esta ayuda:

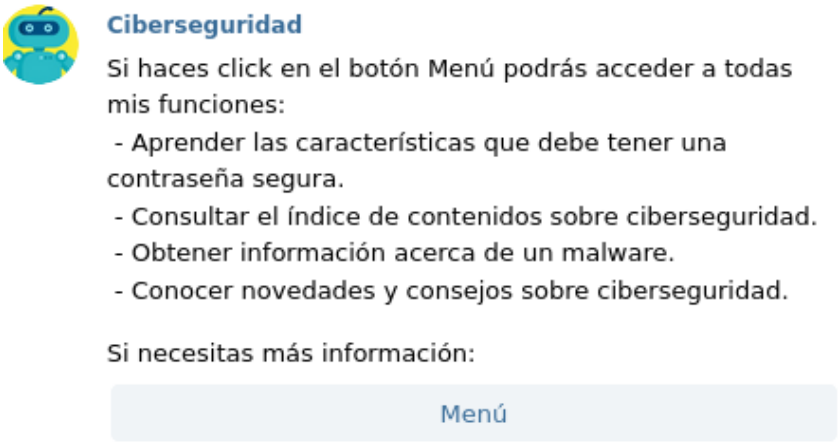


Imagen 32. Prueba con Telegram. Captura de pantalla.

- El usuario rellena un formulario para que el chatbot pueda aconsejarlo [29].

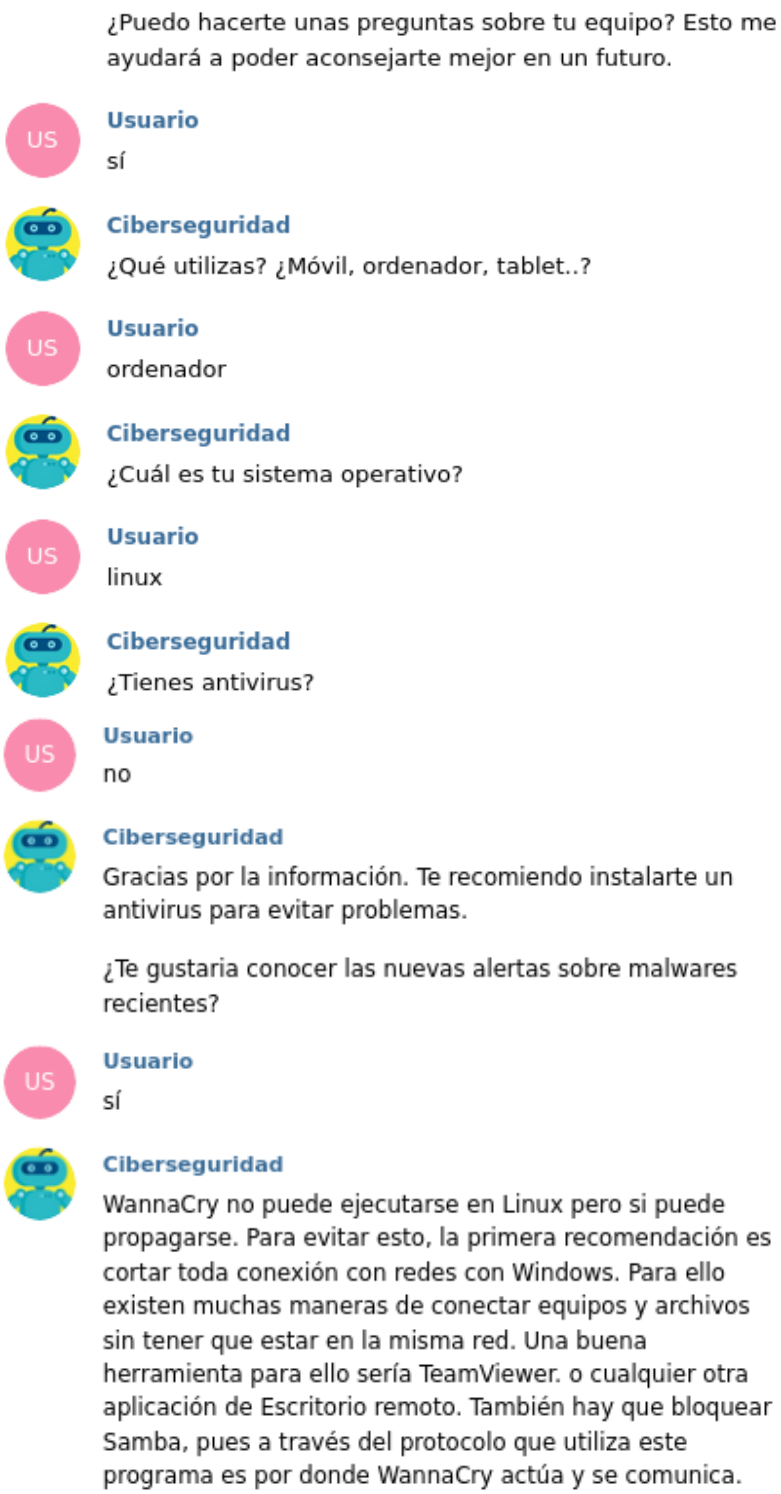


Imagen 33. Prueba con Telegram. Captura de pantalla.

- Búsqueda en la tabla *malwaretype* [31].

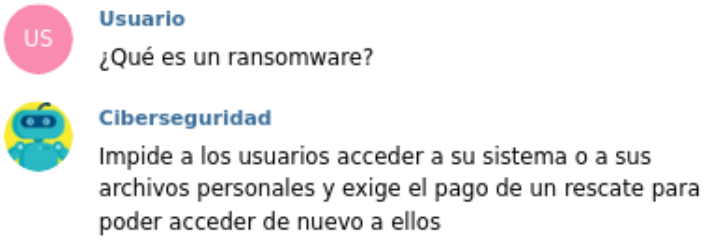


Imagen 34. Prueba con Telegram. Captura de pantalla.

- Le pregunta al usuario cómo solucionó el problema, y almacena la solución propuesta en la tabla *solutionusers* [31].

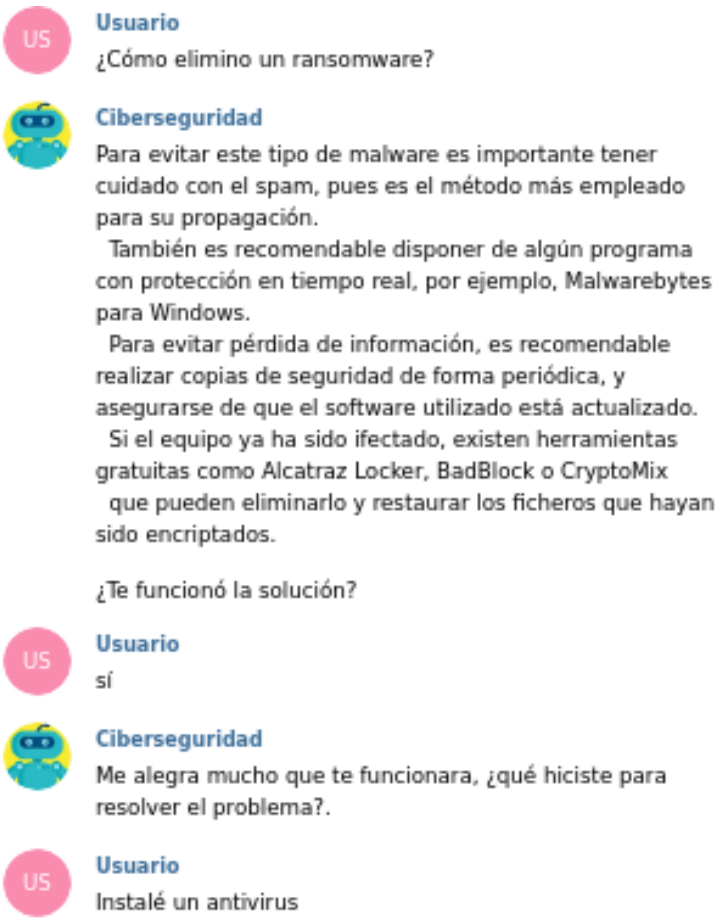


Imagen 35. Prueba con Telegram. Captura de pantalla.

- Si otro usuario volviese a preguntar lo mismo, y a él no le funcionara la primera solución, se le podría recomendar lo propuesto por el usuario anterior.

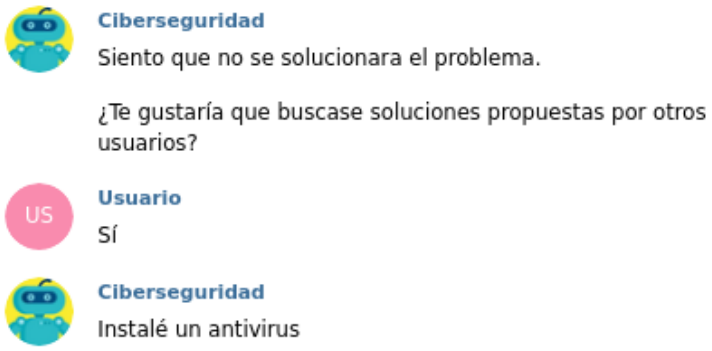


Imagen 36. Prueba con Telegram. Captura de pantalla.

- Evaluar la seguridad de una contraseña introducida por el usuario.

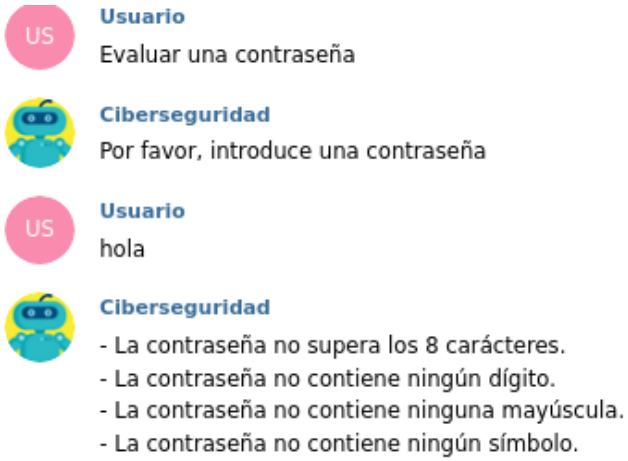


Imagen 37. Prueba con Telegram. Captura de pantalla.

- Generar una contraseña segura y aleatoria, de un tamaño definido por el usuario.

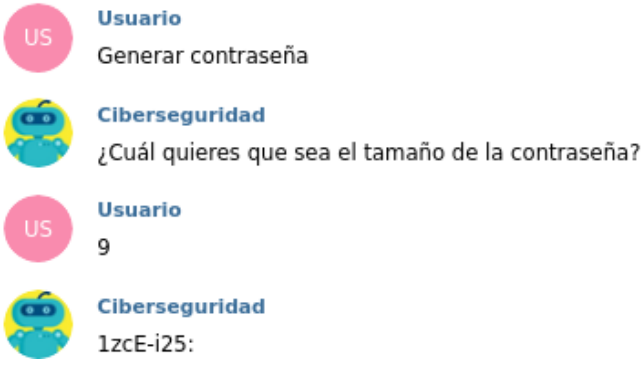


Imagen 38. Prueba con Telegram. Captura de pantalla.

- Si el usuario pregunta sobre algún tema, se le muestran resultados relacionados [25]:

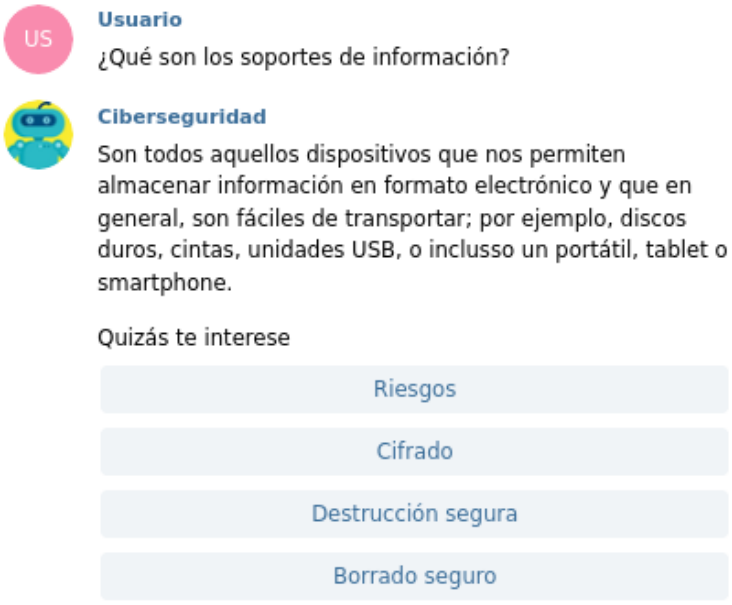


Imagen 39. Prueba con Telegram. Captura de pantalla.

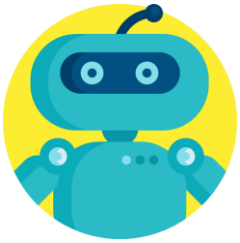


Imagen 40. Icono realizado por Freepik. flaticon.com.

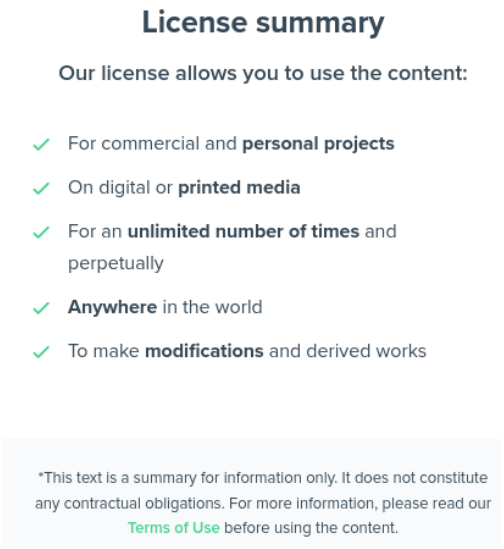


Imagen 41. Licencia del icono utilizado, flaticon.com.

6 Conclusiones

El objetivo principal del TFG es la construcción de un chatbot sobre ciberseguridad, que sea capaz de proporcionar información útil al usuario.

En primer lugar, se pensó qué funcionalidades podría proporcionar un chatbot a diferencia de una guía de usuario, por qué era necesaria su implementación. Como ya se explicó anteriormente, un chatbot puede manejar una gran cantidad de datos, proporcionar respuestas inmediatas, así como analizar y tener en cuenta las propuestas y datos aportados por los usuarios.

En segundo lugar, se realizó una enumeración de las tareas que podía realizar el chatbot, especificando la entrada, salida y los posibles escenarios que se pudieran dar. También se tuvo en cuenta cuáles iban a ser las posibles intenciones del usuario, es decir, cómo el usuario accedería a la información. Al principio, se mantuvo una mecánica de preguntas libres, después, para poder guiar mejor al usuario, se añadió un menú capaz de orientar al usuario durante todo el proceso.

También se pensó en la forma de conocer la opinión del usuario. Así se diseñaron escenarios en los que se recogiesen distintas propuestas de los usuarios, o cómo de efectiva era una solución propuesta a un problema. De esta manera es posible mejorar la herramienta y los datos utilizados por el chatbot.

Una vez definidos los distintos escenarios y funcionalidades, se procedió a realizar un diseño de la base de datos, y de los módulos y acciones que se implementarían en el proyecto.

Durante el desarrollo de este TFG, he podido aprender distintas herramientas que existen a la hora de crear chatbots, así como las diferencias entre base de datos como PostgreSQL y SQLite y en qué momento es recomendable utilizarlas. Una herramienta interesante, es la ejecución del chatbot con *debug* que ofrece Rasa, no solo es útil para detectar errores, también para conocer mejor el proceso: la transformación de cadenas de texto introducidas por el usuario y cómo el sistema predice la siguiente acción a ejecutar.

La principal limitación del chatbot actualmente es no contar aún con las suficientes definiciones y material en la base de datos. Además, actualmente se necesita conocimientos previos sobre SQL para poder actualizar la información proporcionada. En un futuro, se podría mejorar el sistema desarrollando alguna aplicación para poder gestionar los datos sin ser necesarios estos conocimientos.

Referencias

[1] SAS: Software y soluciones de analítica. “Prodesamiento del lenguaje natural. Qué es y por qué es importante”.

[2] Definiciones según la RAE.

[3] Wikipedia. “Procesamiento de lenguajes naturales”, 16 de marzo de 2020 (fecha de última edición).

[4] Digital guide IONOS. “¿Qué es el Natural Language Processing?”.

[5] Patricia Durán. “Qué es DialogFlow y cuáles son sus nuevas actualizaciones en fase beta”, 2 de agosto de 2018.

[6] Devashish Datt Mamgain. “Dialogflow vs Rasa — Which One to Choose?”, 21 de marzo de 2019.

[7] Devashish Mamgain. “Dialogflow vs Rasa – Which One to Choose?”, 28 de Agosto de 2018.

[8] Abhishek Shanbhag. “Comparing The Top Bot Development Frameworks”,

[9] Adnan Rehan. “9 Best Chatbot Development Frameworks to Build Powerful Bots”, 13 de julio de 2019.

[10] Discover.bot. “Building Bots with Wit.ai”, 16 de julio de 2019.

[11] Discover.bot. “Building Bots with Amazon Lex”, 16 de abril de 2019.

[12] Jay Nath. **“**How to Identify the Right Platform for Your Chatbot”, 27 de febrero de 2018.

[13] TablePlus. “SQLite vs PostgreSQL - Which database to use and why?”, 30 de agosto de 2018.

[14] Wikipedia. “SQLite”, 11 de abril de 2020 (última fecha de edición).

[15] Asaf Yigal. “Sqlite vs. MySQL vs. PostgreSQL: A Comparison of Relational Databases”, 8 de noviembre de 2018.

[16] Wikipedia. “PostgreSQL”, 8 de abril de 2020 (última fecha de edición).

[17] Rubén Velasco. ”DB Browser for SQLite, la forma más fácil de crear y editar bases de datos SQLite”, 30 de junio de 2018.

[18] IES Virgen del Espino. “pgAdmin III”.

[19] PYnative. “Python PostgreSQL Tutorial Using Psycopg2”, 11 de abril de 2020 (última fecha de edición).

[20] Documentación de Rasa. “Retrieve a conversations tracker”.

[21] Documentación de Rasa.“Slots”.

[22] Documentación de Rasa. “Domains”.

[23] Documentación de Rasa. ”Actions”.

[24] Documentación de Rasa. “Choosing a Pipeline”.

[25] incibe, Instituto Nacional de Ciberseguridad. “La información”, “El puesto de trabajo”, “Los Dispositivos Móviles”, “Los soportes de información”.

[26] Ricardo Ortín Tomas. “Porque es importante la ciberseguridad. 5 razones”, 4 de octubre de 2018.

[27] John Snow. “Top 5 de los ciberataques más memorables”, 7 de noviembre de 2018.

[28] Mayur Pethani. “Making of Chatbot using Rasa NLU & Rasa Core-Part 2”, 7 de noviembre de 2019.

[29] Joaquín García. “Cómo proteger aún más mi Linux frente a WannaCry”.

[30] Olivia Morelli. “Eliminar el virus WannaCry (Guía de eliminación)”, actualización de agosto de 2017.

[31] Malwarebytes.com. “Todo acerca del ransomware”.

[32] rasa.com

[33] dialogflow.com

[34] dev.botframework.com

[35] wit.ai

[36] aws.amazon.com/es/lex/

[37] slack.com/intl/es-es/

[38] facebook.com

[39] messenger.com

[40] telegram.org

[41] skype.com/es/

[42] azure.microsoft.com/es-es/services/bot-service/

[43] dev.botframework.com

[44] sqlite.org/index.html

[45] postgresql.org

[46] sqlitebrowser.org

[47] pgadmin.org

[48] pypi.org/project/psycopg2/

[49] pypi.org/project/pg8000/

[50] pypi.org/project/py-postgresql/

[51] pygresql.org

[52] sqlalchemy.org

[53] ngrok.com

Glosario

ACID	Atomicidad, Consistencia, Aislamiento y Durabilidad.
IOT	Internet de las cosas.
RDBMS	Sistema de gestión de bases de datos relacionales.
Cracker	Persona que vulnera un sistema de seguridad de forma ilícita.
Ransomware	Malware que bloquea los archivos o dispositivos del usuario.
Gusano	Programa que realiza copias de sí mismo, alojándolas en diferentes ubicaciones del ordenador.
ASR	Reconocimiento Automático de la Voz.
SSL	Secure Sockets Layer. Se instala en el servidor web, permitiendo una conexión segura entre el servidor y un navegador web.
Endpoint	Dispositivo informático remoto que se comunica con una red a la que está conectado.

Anexos

A Manual de instalación

1- Configuración base de datos

A la hora de realizar la conexión a la base de datos (*Database.py*), se debe indicar el usuario, contraseña, nombre de la base de datos y puerto. La creación de las tablas y los *inserts* se encuentran en *BotDB.sql*.

2- Configuración de url con Ngrok

En una terminal se ejecuta el siguiente comando:

/ngrok http 5005

En esta ocasión se ha elegido utilizar el puerto 5005, si utilizásemos otro se debería modificar el Makefile.

```
telegram:
  rasa train
  rasa run -m models -p 5005 --credentials credentials.yml
```

En credentials.yml se modifica el siguiente código:

```
telegram:
  access_token: "971497924:AAHhUDZnfJjIU8so7xMId2c-Mqg-
d4BwQOY"
  verify: "prueba117bot"
  webhook_url: "https://c8340451.ngrok.io/webhooks/telegram/webhook"
```

Access_token y *verify* son parámetros proporcionados por Telegram a la hora de crear el chatbot.

Webhook_url se forma con la url proporcionada por Ngrok:
/webhooks/telegram/webhook

3- Otros cambios

Por defecto, en telegram.py aparece button_type= "inline", se puede modificar a "vertical" para que los botones se muestren de forma vertical, así se consigue que el texto del botón aparezca completo.

4- Despliegue

Después de haber configurado la url y base de datos. Ejecutamos *make action-server* y *make telegram* en diferentes terminales.